CSCE 435 Parallel Computing, Fall 2017
Project 2: parallel sorting in OpenMP

Tim Davis

Assigned Tues, Oct 17. Due at 2pm on Thurs, Oct 26

In this project you will write two parallel sorting methods: a parallel quicksort and a parallel bucket sort, and compare their performance with their sequential verions and with each other

1 Part 1: quicksort

I have written a sequential quicksort `qsort1.c`. It doesn't use OpenMP, except for the timing routing `omp_get_wtime`. Make a copy of this file as `qsort2.c` and parallelize the quicksort.

Don't try to parallelize the partition method. Instead, use the `task` pragma to parallelize the two independent recursive calls to quicksort, for each subproblem. You will need to use several pragmas to do this.

- `#pragma omp task` for each recursive call inside the recursive quicksort routine, `quicksort_recursive`.
- `#pragma omp parallel` and `#pragma omp single no wait`: the top-level of the non-recursive `quicksort` function needs to use these pragmas to call the recursive function. The `single nowait` option means that each subsequent task is done by a single thread.

Also, modify the recursive quicksort routine so that it selects between two sequential calls (not parallel) or two parallel calls, depending on the size of the list. If the list is smaller than some constant size, do not use parallelism. Experiment with which constant to use. This strategy is like the use of insertion sort when `n` is less than 20. Don't modify that part of the code. Instead, do insertion sort if `n<20`, do a sequential recursive quicksort of `n < (some bigger constant)` and do a parallel quicksort otherwise. Pick a good value of this "some bigger constant" that gives you good performance overall.

Also parallelise the creation of the input array `A` and the test to see if the result is sorted. In addition, in your parallel code `qsort2.c`, use the purely sequential sort from `qsort1.c` to sort a copy of the array. Use this to compare the output of your parallel sort to make sure you got the right result. The timing for these two sorts will give your parallel speedup.

Analyze the performance of your parallel quicksort with different values of `n` (say 1 million, 10 million, and 100 million) and different numbers of threads (say 1, 2, 4, 8, 16, and 20). Plot the speedup for each three problem sizes, as a function of the number of threads.

2 Part 2: bucket sort

The second method is a parallel bucket sort. Write an entirely new program to do this (feel free to borrow from `qsort1.c` of course, like the main program). Call it `bucket.c`. In this method, assume the `double` array `A` you are sorting contains integers in the range `0` to `K-1`, for some given integer `K`. (Make `K` an input to the program via `argc` and `argv`). Assume `K` is small (no larger than 2^{20}), but keep `A` as `double` for a fairer comparison with the quicksort methods.

The outline of the bucksort algorithm with `p` threads is as follows. Each step is fully parallel unless described as sequential.

```
step 1: allocate a 2D Count array of size p-by-K and set it to zero
step 2: each thread iterates through part of the array A and counts
    how many times it sees each value; that is, it does Count[id][A[i]]++
    where id is the thread id
step 3: sum up the Counts for each processor
step 4: sequentially go through the Count array and compute its prefix sum.
    This takes only K iterations and it will be hard to get good parallel
    performance for this step.  So just do it sequentially.
step 5: recreate the A array from the summed up Count
```

Analyze the performance of your bucket sort for the same values of `n` and same set of threads as part 1, and also with two values of `K`. Try `K` of 1024 and 2^{20}.

3 Files provided

- `Makefile`: you will need to modify this; it just does `qsort1.c` for now.
- `qsort1.c`: the sequential quicksort. Leave this unchanged, but use it to help you write your `qsort2.c` and `bucket.c`. Note it uses `omp_get_wtime` instead of the tic/toc functions from project 1.

4 What to turn in

Write up a project report discussing your solutions and the performance obtained by each method. Compare the sequential and parallel quicksort. Also compare your parallel quicksort with your parallel bucketsort. They won't be sorting the same kind of data, but both will be sorting `double` arrays `A` of the same size. Which method is faster, and does it depend on `K`?

Also turn in all your codes, including the files I have provided you. Include all this and your project writeup (as PDF) in a single zip file.

Honesty statement: As Aggies, we follow the Aggie Honor Code. Period. This is a solo assignment. Do not share code; I can find it no matter how you obscure it. Trust me on that; I have been writing code, now used by millions, since before you were born. Copying code results in a zero on the project and a record in the honor system; don't risk it.